

# Evaluation of Parsons Problems with Menu-Based Self-Explanation Prompts in a Mobile Python Tutor

**Geela Venise Firmalo Fabic**, *Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand*

**Antonija Mitrovic**, *Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand*

**Kourosh Neshatian**, *Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand*

**Abstract.** The overarching goal of our project is to design effective learning activities for PyKinetic, a smartphone Python tutor. In this paper, we present a study using a variant of Parsons problems we designed for PyKinetic. Parsons problems contain randomized code which needs to be re-ordered to produce the desired effect. In our variant of Parsons problems, students were asked to complete the missing part(s) of some lines of code (LOCs), and rearrange the LOCs to match the problem description. In addition, we added menu-based Self-Explanation (SE) prompts. Students were asked to self-explain concepts related to incomplete LOCs they solved. Our hypotheses were: (H1) PyKinetic would be successful in supporting learning; (H2) menu-based SE prompts would result in further learning benefits; (H3) students with low prior knowledge (LP) would learn more from our Parsons problems in comparison to those with high prior knowledge (HP). We found that the participants' scores on the post-test improved, thus showing evidence of learning in PyKinetic. The experimental group participants, who had SE prompts, showed improved learning in comparison to the control group. Further analyses revealed that LP students improved more than HP students and the improvement is even more pronounced for LP learners who self-explained. The contributions of our work are a) pedagogically-guided design of Parsons problems with SE prompts used on smartphones, b) showing that our Parsons problems are effective in supporting learning and c) our Parsons problems with SE prompts are especially effective for students with low prior knowledge.

**Keywords.** Mobile Python tutor, menu-based self-explanation, Parsons problems

## INTRODUCTION

Understanding programming concepts and acquiring the skill of code writing are both essential in learning programming. Novices struggle with problem solving due to the lack of declarative and/or procedural knowledge (Anderson, 1982). In programming, declarative knowledge includes the syntax of the programming language, while procedural knowledge comprises of writing code constructs into a meaningful working program. The goal of our project is to support novices in learning Python by having simple activities that support acquisition of both conceptual and procedural knowledge. We present PyKinetic, a Python tutor for Android smartphones which is designed for novices and aimed as a complement to traditional lectures (Fabic, Mitrovic and Neshatian 2016a; 2016b). Mobile learning is proven effective in many domains including computing education (Hürst, Lauer, and Nold, 2007; Karavirta, Helminen, and Ihantola, 2012; Boticki et al., 2013; Vinay, Vaseekharan and Mohamedally, 2013; Wen and Zhang, 2015; Mbogo, Blake, and Suleman, 2016; Grandl et al., 2018; Oyelere et al., 2018). However, there are only a few mobile applications in the literature focusing on enhancing programming skills through control-flow learning and code writing (Karavirta, Helminen, and Ihantola, 2012; Vinay, Vaseekharan and Mohamedally, 2013; Grandl et al., 2018; Mbogo, Blake, and Suleman, 2016). Therefore, there are opportunities to contribute to literature by investigating a mobile tutor focused on enhancing coding skills.

For the first version of PyKinetic, we designed a variant of Parsons problems (Parsons and Haden 2006), which are exercises consisting of a set of randomized lines of code (LOCs) that need to be rearranged in the correct order to produce a desired outcome. Parsons problems are suitable for novices as they contain correct syntactic constructs and impose low cognitive load (Morrison et al. 2016). Furthermore, learners often lack mental models and have difficulty in translating a problem into manageable tasks (Winslow, 1996). Parsons problems provide scaffolding helpful for novices, unlike in traditional code-writing exercises, where a student is expected to construct code where the only scaffolding is the problem description. Previous research found Parsons problems to have a moderate positive correlation with code writing (Denny et al., 2008). In order to further target code writing skills, our implementation of Parsons problems includes incomplete lines of code, a variant of Parsons problems (Ihantola, Helminen and Karavirta 2013; Fabic, Mitrovic and Neshatian 2017b). The incomplete LOCs within the Parsons problems act like micro-scaffolded code writing exercises. Therefore, in addition to rearranging LOCs, learners are required to fill in missing code elements.

The second reason for choosing Parsons problems is their compatibility with the smartphone interface. Parsons problems implemented on computers and mobile devices are solved by drag and drop movements (Parsons and Haden 2006; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013; Harms, Chen, and Kelleher 2016; Kumar, 2018). The drag and drop motion used in solving Parsons problems is commonly used when interacting with smartphones. Moreover, Parsons problems can be implemented in smartphones without having to sacrifice too much screen space as they do not require typing. Requiring the student to write code on a smartphone might not be ideal, as half of the screen will be obstructed by the keyboard when typing. Karavirta et al. (2012) also perceived Parsons problems to be suitable for mobile device interface and developed MobileParsons for Android and iOS.

Parsons problems with incomplete LOCs gave us a good starting point in targeting procedural knowledge. In order to support the acquisition of conceptual knowledge, we introduced *self-explanation* (SE) prompts. Self-explanation was first introduced as open-ended questions requiring the learner to explain presented learning material to oneself, to gain deeper understanding (Chi et al. 1989). Previous research provides evidence that self-explaining enhances conceptual knowledge (Najar, Mitrovic and McLaren, 2016). Several types of SE prompts are reported in literature (Wylie and Chi, 2014), including menu-based SE prompts. We designed menu-based SE prompts for PyKinetic as previous research shows that menu-based SE prompts are more effective than open-ended SE prompts (Johnson and Mayer, 2010; Gadgil, Nokes-Malach and Chi, 2012; van der Meij and de Jong, 2011). Moreover, using menu-based SE prompts does not require the learner to type, which is better suited to smartphones.

In this paper, we present an evaluation study which investigates the effectiveness of our Parsons problems, especially for students with low prior knowledge. By including SE prompts, we address a research gap in the SE literature. Prior work on SE used either worked examples, e.g. (Berthold, Eysink and Renkl 2009; Najar, Mitrovic and McLaren, 2016), or problem solving (unsupported or tutored), e.g. (Kwon, Kumalasari and Howland 2011). In our learning activities, students are rearranging LOCs and completing some of them; such activities are harder than studying worked examples, but less demanding than problem solving. We therefore wanted to determine the influence of SE on learning in this context.

The study included two groups of programming students learning with PyKinetic, in which the experimental group used a version of PyKinetic with SE prompts, and the control group had the same Parsons problems without SE. Such experimental design (with and without SE) is like other experiments reported in literature (O’Neil et al., 2014; Hsu et al. 2012; Rau, Aleven and Rummel 2015; Aleven and Koedinger 2002). We had three hypotheses for this study: (H1) PyKinetic would be successful in supporting learning; (H2) SE prompts would result in further learning benefits; and (H3) low prior knowledge (LP) students would learn more from our Parsons problems in comparison to high prior knowledge (HP) students. Most implementations of Parsons problems are developed for personal computers apart from MobileParsons (Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013). Therefore, the main goal of our study was to investigate the effectiveness of Parsons problems on smartphones when combined with SE prompts.

One of the main contributions of this work include a pedagogically-guided design of Parsons problems combined with SE prompts, which differs from common implementations in four ways. We emphasize that our

main contribution is not the individual features of our Parsons problems, but rather our design as a whole. The combination of our Parsons problems enabled us to offer an effective learning activity for supporting learning programming on smartphones, especially for students with low prior knowledge. Firstly, in our design students solve Parsons problems in the same area where the LOCs are presented. Therefore, there is no separation at all between the blocks of code in the problem and the student’s solution. We propose that combining the two areas makes better use of the limited screen resource in smartphones. Secondly, in our Parsons problems all lines are provided with the correct indentation which are necessary to mark the start and end statements in Python. We decided to provide LOCs with correct indentation because we are focusing on novice learners; correctly indented lines provide additional scaffolding to learners. Thirdly, all blocks of code contain only single LOCs, like work by Garner (2007) and Kumar (2018). Moving blocks of LOCs could potentially prevent the student from thinking about individual LOCs; for that reason, we require the student to move single LOCs. Finally, our implementation contains incomplete LOCs with accompanying menu-based SE prompts that learners must answer upon solving the incomplete LOCs. To the best of our knowledge, we were the first ones to integrate Parsons problems with SE prompts. We have chosen to specifically use menu-based SE prompts as they are easier to complete on smaller screened devices. Furthermore, our other contribution is designing and evaluating SE on smartphones which has also not been done before to the best of our knowledge.

In the next section, we present prior research done on Parsons problems and self-explanation. We then introduce PyKinetic, followed by sections presenting the experiment design and the results. We end the paper with a discussion of the results, conclusions and future work.

## RELATED WORK

### Parsons problems

Parsons problems or Parsons puzzles were originally proposed as a fun way for introductory learners of Turbo Pascal to improve their problem-solving skills (Parsons and Haden 2006). Parsons problems usually contain a problem area and a solution area. The problem area contains blocks of code, where some blocks may contain more than one line of code. The learner solves the problem by dragging blocks from the problem area onto the solution area. Most often Parsons problems are implemented for personal computers, where screen space is not an issue (Parsons and Haden 2006; Garner, 2007; Ihantola and Karavirta 2011; Harms, Chen, and Kelleher 2016; Ericson, Margulieux and Rick 2017; Kumar, 2018; Hosseini, 2018). Despite restrictions on screen size, MobileParsons implemented Parsons problems on mobile devices with two work spaces; dragging and dropping blocks of code side by side when on landscape mode, and bottom to top when on portrait mode (Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013). Variations of Parsons problems include extra incorrect LOCs (*distractors*) (Parsons and Haden 2006; Garner, 2007; Denny et al., 2008; Harms, Chen, and Kelleher 2016; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Kumar, 2018), or incomplete LOCs which require the learner to provide missing elements (Ihantola, Helminen and Karavirta 2013; Fabic, Mitrovic and Neshatian 2017b).

Some researchers find the complexity of Parsons problems to be in between code tracing and code writing (Lister et al. 2010), while others believe Parsons problems to be easier than code tracing (Lopez et al. 2008). Denny et al. (2008) explored the potential of Parsons problems as exam questions and found a weak correlation between scores on Parsons problems with code tracing questions, and a moderate positive correlation with code writing. A recent study was conducted using Parsons problems as an assessment activity, as they pose a lower cognitive load compared to traditional code writing (Morrison et al. 2016). The authors propose that this is because learners are only required to focus on sequencing the code, since the correct syntactic constructs are provided. However, this may not be always true, as Parsons problems may require higher cognitive load depending on the complexity, type, and interface used (on paper or on a device). Moreover, others also perceive that the position of Parsons problems in the hierarchy of programming skills depend on complexity and type of Parsons problems (Ihantola

and Karavirta 2011). More factors that could be considered are scaffolding and feedback provided. Recent work by Ericson et al. (2017) compared learning gains between three groups of students solving the same set of Python problems, but with each group using a different method of problem solving. One group was solving two-dimensional Parsons problems with distractors, another was tasked to fix the broken code, and the third group wrote the same code without scaffolding. Two-dimensional Parsons problems required learners to specify the correct indentations for each LOC in addition to re-ordering the LOCs (Ihantola and Karavirta 2011). All three groups showed evidence of learning from pre- to post-test on fixing and writing code, which is consistent with the positive correlation between Parsons problems and code writing found by Denny et al. (2008). Furthermore, Ericson and colleagues found neither a significant difference between the three groups on their learning performance, nor a significant difference on retention of knowledge when evaluated a week after the study. A notable finding was that the group who solved two-dimensional Parsons problems with distractors were significantly faster than the other groups who solved the same set of problems but used a different method of problem solving. Therefore, the authors found the Parsons problems group to be most efficient as they gained the same knowledge significantly faster than the other two groups.

## Self-explanation

Self-explanation is a learning activity in which the learner is explaining the learning material (e.g. a worked example or instructional text) to oneself, by making inferences from existing knowledge (Chi et al. 1989). SE results in deep learning, as it allows the learner to integrate new with existing knowledge, identify and eliminate misconceptions, and reflect on their knowledge (Chi 2000). SE has been shown to improve problem-solving skills and knowledge on domain principles when learning from worked examples (Chi et al. 1989). When self-explaining and learning from text, conceptual knowledge was also developed (Chi et al., 1994). Furthermore, Ferguson-Hessler and de Jong (1990) found that self-explaining supports conceptual knowledge more than procedural knowledge, regardless of the abilities of the student.

SE prompts were first introduced as open-ended questions which encourage learners to think without any set limitations. Wylie and Chi (2014) discuss the range of SE prompts that have emerged, arranged by increasing amount of support provided: open-ended, focused, scaffolded, resource-based and menu-based prompts. *Focused SE prompts* are variations of open-ended prompts, which provide more explicit instructions (i.e. compare and contrast). *Scaffolded SE prompts* provide explanations with missing keywords to be filled in by the learner. *Resource-based SE prompts* offer a resource library (such as a glossary) which students can refer to. Lastly, *menu-based SE prompts* are similar to resource-based self-explanation prompts, but selections are provided from a menu instead of a resource library. A study by Aleven et al. (2004) compared open-ended SE with resource-based SE, where students selected reasons for their actions from a glossary. The results revealed no significant differences with overall learning gains between open-ended group and resource-based group, showing that open-ended SE is not always the best. However, authors suggest that the benefits of open-ended SE possibly did not manifest since it took more time to self-explain with their own words than selecting an explanation.

Johnson and Mayer (2010) compared open-ended with menu-based SE prompts in a game-like environment about electrical circuits. Johnson and Mayer first conducted an experiment comparing transfer tests scores of participants who used menu-based SE prompts vs. participants without any SE prompts. The results revealed that the SE group significantly outperformed the group without self-explanation. The second experiment compared transfer test performances of the menu-based SE group and no self-explanation group from the first experiment, with a new group using open-ended SE prompts. Results showed that learners who used open-ended SE had similar transfer test results to learners who did not do any self-explanation. Therefore, the authors found that menu-based SE were more effective than open-ended SE prompts. The authors explained the results by reflecting on the limited cognitive capacity of learners based on the Cognitive Load Theory (Plass, Moreno, Brünken 2010). Within a game-like environment, menu-based SE prompts may have resulted in effective learning due to minimizing extraneous cognitive load while fostering germane and intrinsic load. Johnson and Mayer (2010) concluded that menu-based SE prompts may be more suitable in complex environments compared to open-based prompts. Menu-based SE

prompts were found more effective than open-ended prompts in several other studies (Gadgil, Nokes-Malach and Chi, 2012; van der Meij and de Jong, 2011)

Wylie and Chi (2014) advise that regardless of the form, all SE prompts may lead to deeper learning. Further insights into the effectiveness of SE prompts can be obtained within the ICAP (Interactive, Constructive, Active and Passive) framework, which focuses on learners' engagement (Chi and Wylie 2014). According to the ICAP framework, learning increases as engagement increases. When referring to the ICAP framework, self-explanation is constructive in nature. However, as mentioned previously from work by Aleven et al. (2004), open-ended SE prompts, which may be considered more constructive than other types, were not proved to be more effective than menu-based SE prompts. Therefore, various aspects need to be considered when choosing the type of SE prompts to use. For example, an important consideration in designing an SE prompt, regardless of its form, is providing appropriate feedback. Learners especially novices have misconceptions, which require guidance via feedback. If feedback is not properly given, learners continue to operate using their false understanding which could be detrimental to their learning. However, an intricate balance needs to be achieved to help the learners just enough to facilitate in enhancing their skills rather than just spoon-feeding information.

## **PYKINETIC**

PyKinetic is a mobile tutor developed using Android SDK to teach Python 3.x programming. We developed the first prototype of PyKinetic in 2015 (used in the pilot study) The version of PyKinetic used in the study and presented in this paper supports Android devices with version 4.0 (Ice Cream Sandwich) and higher.

PyKinetic presents a Parsons problem by first showing the problem statement in a dialog box, together with the expected output. LOCs are presented in a random order, with correct indentation. The learner completes problems by dragging and dropping LOCs using the drag handlers on the left-hand side of each line. A problem is completed if the LOCs are in the correct order matching the desired outcome as described in the problem statement.

### **Pilot study**

We conducted a pilot study (Fabic, Mitrovic and Neshatian 2016b) with a prototype of PyKinetic containing 24 Parsons problems with and without distractors. All LOCs were complete (without any missing keywords). Distractors were extra incorrect LOCs that were not part of the solution. The prototype covered the following eight Python topics: string manipulation, conditional statements, lists, for loops, while loops, dictionaries, tuples, and data types. There were three Parsons problems for each topic. All topics had one Parsons problem without distractors and two with distractors. For example, for string manipulation there were three Parsons problems: one without distractors and the other two with distractors. The problems had between 3 to 16 LOCs, with a maximum of 5 distractors. To eliminate a distractor, students tapped on the 'X' icon shown on the right-hand side of each LOC, to delete that LOC. Learners retrieved LOCs that had been deleted by tapping on the trash icon and selecting the desired LOCs.

The study had one-hour long controlled sessions, undertaken individually. The participants were eight students and five tutors, volunteers from an introductory programming course at University of Canterbury. The participants were instructed to follow the think-aloud protocol (Somerén, Barnard and Sandberg 1994), and to attempt at least one problem from each topic. The goals of the pilot study were to evaluate the usability of PyKinetic, and identify sub-optimal strategies used by students compared to those demonstrated by tutors. We recorded the screen of the smartphones while the participants were solving problems. As expected, the pilot study revealed that the tutors solved the problems more efficiently than the students. The observations showed that less complex problems presented in the landscape mode appeared to be more difficult compared to more complex problems presented in the portrait mode. A possible explanation for these observations was that participants could only see half of the LOCs in the landscape mode, which may have overloaded the working memory of the participants; as they could only see half of the code at a time. As the result of the findings from the pilot study, we

enhanced the aesthetics, and added additional description for complex problems. Furthermore, all problems in PyKinetic are now presented in the portrait mode.

## Full evaluation study

For the full evaluation study, we developed a new version of PyKinetic containing a different variant of Parsons problems. Instead of using distractors, the version of PyKinetic used in the study contained Parsons problems with incomplete LOCs. Solving Parsons problems with incomplete LOCs may be closer to enhancing code writing skills than Parsons problems with/without distractors. Furthermore, a recent study found that Parsons problems with distractors decrease learning efficiency of middle-school children aged 10-15 (Harms, Chen, and Kelleher 2016).

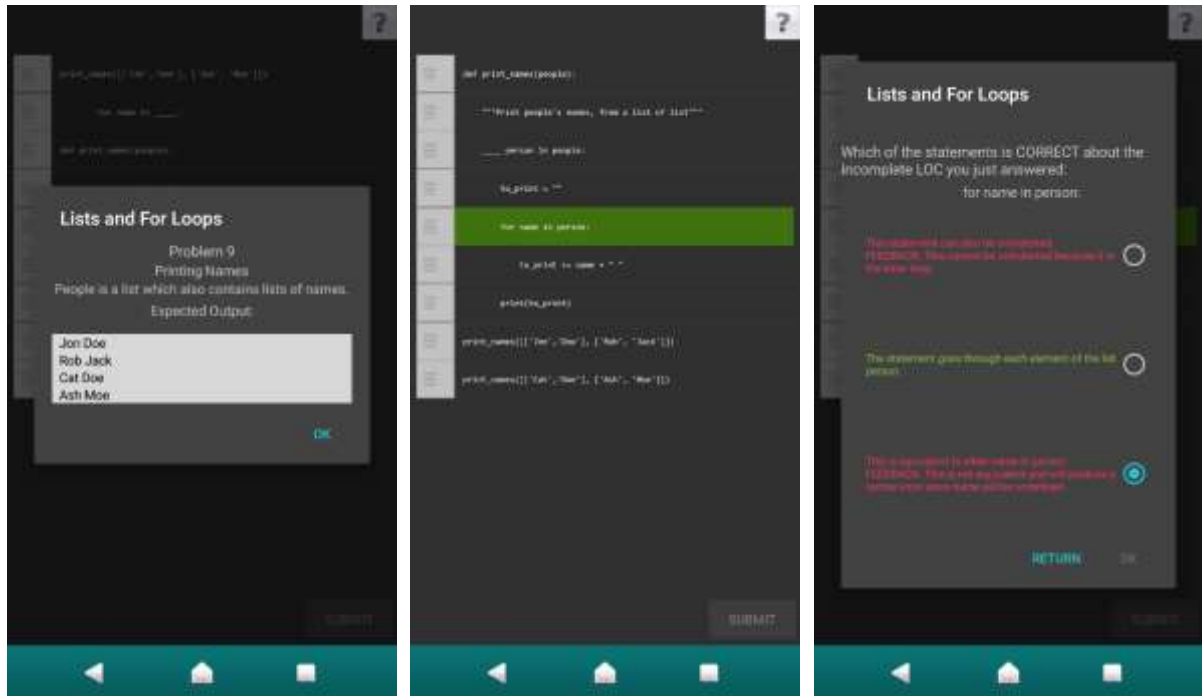
For the study, we implemented two versions of PyKinetic with incomplete LOCs that are identical in every way (i.e. design, control, and set of Parsons problems), apart from the additional SE prompts offered in one of the versions used by the experimental group. For each version, there were 15 Parsons problems where the first two problems were used for practice. The remaining 13 problems had between 3 and 16 LOCs, with a maximum of 3 incomplete LOCs. There were six Python topics covered: string manipulation, conditional statements, while loops, for loops, lists, and tuples. Problems were given in a fixed order of increasing difficulty. A problem must be completed before proceeding to the next problem. The first half of the problems focused on a single topic, while the other problems covered at least two topics each. The initial seven problems were code snippets, while the remaining eight problems were functions with function calls. For all problems with functions, there was one function; and function calls were also required to be rearranged to match the problem description and the expected output. PyKinetic recorded information about all actions performed by the participants.

For each problem, the problem details were displayed first, including the expected output and problem description (Figure 1, left). Correct indentations were provided for all LOCs as scaffolding. An incomplete LOC contained a blank space, which may contain more than one keyword. For example, the third LOC of the problem shown in Figure 1 (middle) is: `_____ person in people:.` The alternatives given to fill in the blank space were: a) for each; b) while; and c) for. In this case, the correct answer is c), the incomplete LOC only required one keyword, so the final answer is: `for person in people:.` Another example of an incomplete LOC from a more difficult problem is: `for num in range(_____):` with the alternatives given: a) number-1, 0, -1; b) 1, number; c) 1, number, -1; and d) number, 1. Notice that the blank line is longer, indicating that more keywords are required. The correct answer for this incomplete LOC is a), so the completed LOC is `for num in range(number-1, 0, -1):.` Four incomplete LOCs contained only a blank line each without any code; only the correct indentation was provided as scaffolding. Seven incomplete LOCs had sets of three options each, while the other fifteen incomplete LOCs had a set of four options each. To solve problems which contain more than one incomplete LOC, learners should select an incomplete LOC one at a time by long tapping on the LOC. Then, the learner chooses the correct choice for the blank from a set of provided options, by tapping between alternatives instead of typing, like in (Ihantola, Helminen and Karavirta 2013). Learners submit their solution to an incomplete LOC by long tapping on the LOC.

There were two ways that PyKinetic showed feedback when solving incomplete LOCs: highlighting the LOC to indicate its status change and displaying feedback messages. Feedback on solving incomplete LOCs was only triggered by long tapping on the incomplete LOC either to select it or to submit an answer. The incomplete LOCs were highlighted in three different colors: turquoise, red, and green. Turquoise indicates that the learner has selected the incomplete LOC and intends to solve it. Red indicates that the learner's answer is incorrect. Lastly, green specifies that the learner has selected the correct choice for the LOC. When a learner submits an incorrect answer to an incomplete LOC, PyKinetic highlights the LOC red and shows a feedback message: "[alternative choice here] is incorrect.". If the learner successfully solves an incomplete LOC, it is highlighted in green. Furthermore, the control group receives a message "Correct! Great job!". On the contrary, the experimental group is given the SE prompt which corresponds to the incomplete LOC they solved.

All incomplete LOCs must be completed before the learner can submit their solution to the Parsons problem (by clicking on the Submit button). The Submit button is initially disabled and only activates when all incomplete LOCs are solved. Furthermore, this button is only used to put forward a solution for the entire Parsons problem

(not for submitting an answer to an incomplete LOC). When reordering LOCs in the Parsons problem, feedback is only given upon clicking the Submit button. There were only two feedback messages given when rearranging LOCs in the Parsons problem. Learners receive either: “*Correct! Great job!*” for a completed Parsons problem, or “*Check the order of your solution.*”



**Fig. 1.** Problem description (left); a state of a Parsons Problem with two incomplete LOCs with one LOC already completed, highlighted in green (middle); a conceptual SE prompt given for the completed LOC (right)

To further improve learning, we introduced menu-based self-explanation (SE) prompts, given after the learner finishes an incomplete LOC. There were 22 SE prompts in total; 14 prompts were conceptual questions and 8 were procedural questions. Conceptual questions covered declarative knowledge, assessing syntax and theoretical matters. On the other hand, procedural questions evaluated learners’ understanding of code execution. Conceptual questions can be answered without necessarily reading the entire program, whereas procedural questions require the learner to mentally execute and trace the code. Half of the conceptual questions had three choices each, while the other half had four choices each. On the other hand, two of the procedural questions had three choices each, while the other six had four choices each. The SE prompts were related to the same topics covered by the Parsons problems. However, there were four additional topics covered: assignment statements, variables, print statements, and functions.

**Table 1.** First example of a conceptual SE prompt (shown in Figure 1, right screenshot)

Which of the statements is CORRECT about the incomplete LOC you just answered:  
for name in person:

SE prompt options	Feedback received by learner in each option
The statement can also be unindented.	This cannot be unindented because it is the inner loop.
The statement goes through each element of the list person.	No feedback (only highlighted in green since this is the correct option)
This is equivalent to while name in person:	This is not equivalent and will produce a syntax error since name will be undefined.

The conceptual SE prompt shown in Figure 1 (also presented in Table 1) is given after the learner worked on the incomplete LOC *for name in \_\_\_\_\_*. The learner chose from the alternatives a) people; b) person; c) my\_people and selected the correct option *for name in person*. PyKinetic acknowledges the correct choice for the incomplete LOC by highlighting it in green (Figure 1, middle). The learner next gets the SE prompt, which in this case is related to lists and for loops (Figure 1, right). In this case, the learner is asked to identify the correct statement about the completed LOC, concerning those two Python topics. The learner cannot skip any SE prompts, and has only a single attempt on each prompt. In Figure 1 (right), the learner selected the third option. After the learner submits his/her answer for the SE prompt, PyKinetic shows the correct option in green and incorrect options in red and provides additional feedback for incorrect choices.

The example in Table 1 targets the for-loop statement completed by the learner. The SE prompt is considered as conceptual because it is testing the learner on declarative knowledge about for loops. Specifically, it is asking the learner about the role of indentations in a for loop and its mechanism. Also, it is testing the learner on syntax, and explicitly whether the exact same code would work if replaced with a while loop.

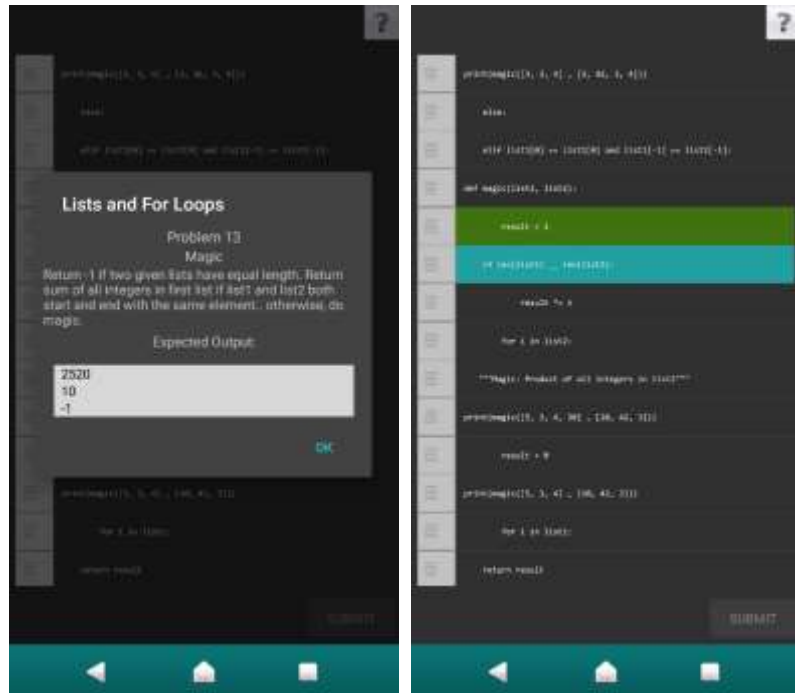
**Table 2.** Second example of a conceptual SE prompt

Select all INCORRECT statements about the incomplete LOC you just answered:  
message = "says hello"

SE prompt options	Feedback received by learner in each option
The variable message has a string value	says hello is in enclosed with quotation marks which makes it a string
message = says hello is equivalent to this statement	No feedback (only highlighted in green since this is one of the correct options)
"says hello" = message is also equivalent to this statement	No feedback (only highlighted in green since this is one of the correct options)

Another example of a conceptual SE prompt is shown in Table 2. This example assesses the learner's knowledge about the syntax of assignment statements, and about string variables. The learner worked on the incomplete LOC *message = \_\_\_\_\_*. Then, the learner successfully selected the correct option *"says hello"*. One thing to note is that the example in Table 2 presents a negatively phrased question as opposed to the example in Table 1. More specifically, the students need to identify the incorrect statements related to the LOC *message = "says hello"*.





**Fig. 2.** Problem description (left); Parsons Problem with two incomplete LOCs, one completed highlighted in green and second selected by the learner and still incomplete highlighted in turquoise (right)

PyKinetic also offers procedural SE prompts, like the one illustrated in Table 3. The LOC highlighted in green in Figure 2 (right) corresponds to a procedural question. This LOC (*result = 1*) started as a blank keyword (\_\_\_\_\_) with indentation provided. The learner chose between alternatives: a) *result = i*; b) *result = list2[i]*; c) *result = 1*; and d) *return result = 0*. As displayed in Figure 2 (right), the learner successfully chose the correct answer c) *result = 1*. Upon completion, PyKinetic gave the learner the SE prompt shown in Table 3. The question presented in Table 3 is explicitly phrased to target the incomplete LOC. The question in Table 3 is procedural because to answer the question, the learner must think about how the code would be executed and understand why this is needed within the context of the expected output. The content of the question in Table 3 aids the reasoning of the learner to understand the role of this LOC within the entire program.

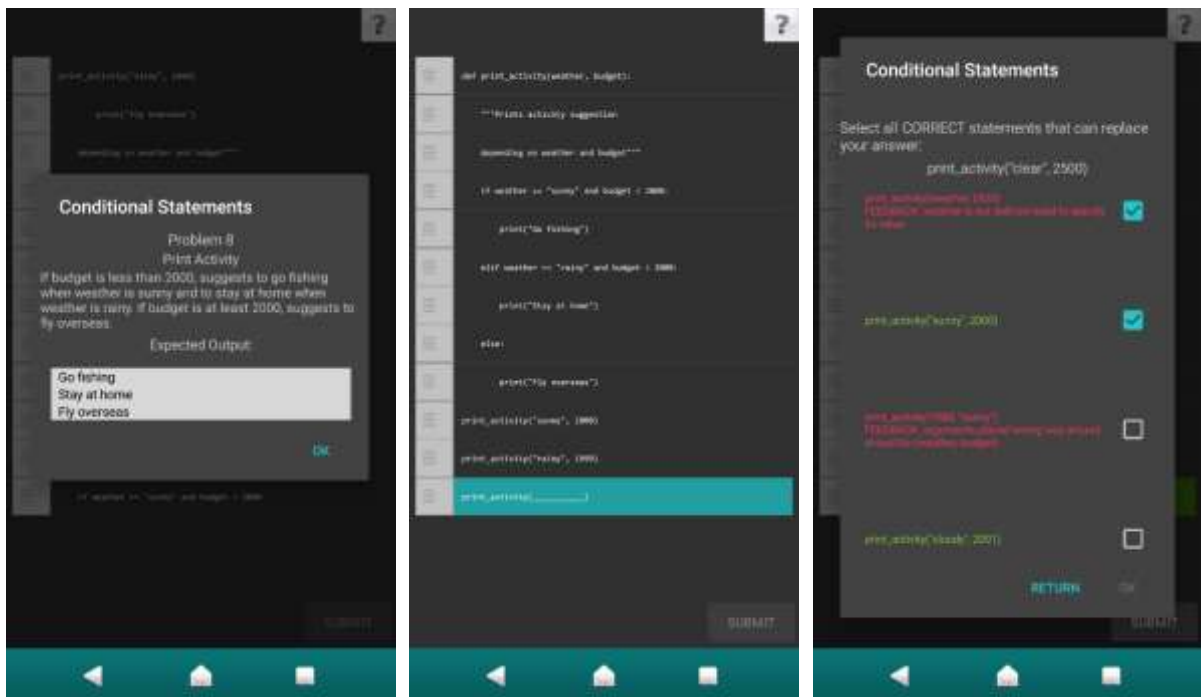
**Table 3.** First example of a procedural SE prompt

Why was this line necessary with a value of 1? Select all that apply:  
*result = 1*

SE prompt options	Feedback received by learner in each option
Magic will work well without this line	This is needed to initialize result
result needs to be initialized before assignment inside the for loop.	No feedback (only highlighted in green since this is one of the correct options)
Initial value of 1 needed for multiplication	No feedback (only highlighted in green since this is one of the correct options)

Figure 3 shows a relatively easier problem on conditional statements which contains one incomplete LOC (highlighted in turquoise in Figure 3, middle). In this example, the learner has correctly rearranged the LOCs in

the problem but have not completed the incomplete LOC. The first two test cases were ordered correctly and displays “Go fishing” and “Stay at home” respectively. In this example, the next goal is to complete the last test case to produce an expected output of “Fly overseas”. The options provided for the incomplete LOC are: a) “clear”, 2500; b) “stormy”, 198; c) “windy”, 1000; or d) “sunny”, 1998. This problem contains two other function calls (i.e. the two LOCs above the highlighted LOC). These function calls output “Go fishing” and “Stay at home” respectively, and the learner had placed them in the correct order. The learner next must complete the function call to output “Fly overseas”. In this example, the correct answer was “clear”, 2500 ; and the completed LOC is `print_activity("clear", 2500)`, shown in (Figure 3, right). After the learner solved the incomplete LOC, the SE prompt is shown. In this case, another procedural SE prompt was given, asking which of the provided statements can replace the incomplete LOC (Figure 3, right; Table 4). This is a procedural prompt because it specifically asks the learner to supply an alternative test case that would produce the same expected output which is “Fly overseas”. To be successful in solving the question, one must mentally execute the code to find out what works for the program. Therefore, relying solely on declarative knowledge about function calls will not be adequate.



**Fig. 3.** Problem description (left); Parsons Problem with one incomplete LOCs selected by the learner, and therefore highlighted in turquoise (middle); a procedural SE prompt given when highlighted LOC is completed (right)

All SE prompts showed the LOC correctly answered by the learner. Most SE prompts (68%) have multiple correct choices provided (as in Figure 3, right), while the others only have a single correct choice. For questions with multiple correct choices, learners were required to identify all correct choices. Furthermore, the SE prompts were phrased in three ways. Some SE prompts were phrased in the positive manner, asking the student to select correct statement(s). Other SE prompts were phrased negatively, requiring the student to select all options which are incorrect. Lastly, the third form were more directly phrased, as in this example: “*In checking for equality in Python using == select all that apply:*” and “*Why was this line necessary with a value of 1? Select all that apply:*”. Feedback “*Correct! Great job!*” is displayed when an SE prompt is answered correctly. If the learner’s response is incorrect, an explanation is shown on all wrong options (Figure 3, right). Regardless of the solution, when the

learner submits their answer, all wrong options in the SE prompt are shown in red font color, while the correct options are shown in green font color (Figure 3, right).

**Table 4.** Second example of a procedural SE prompt (as shown in Figure 3)

Select all CORRECT statements that can replace your answer:

`print_activity("clear", 2500)`

SE prompt options	Feedback received by learner in each option
<code>print_activity(weather, 2930)</code>	weather is not defined need to specify its value
<code>print_activity("sunny", 2000)</code>	No feedback (only highlighted in green since this is one of the correct options)
<code>print_activity(1988, "sunny")</code>	Arguments placed wrong way around should be (weather, budget)
<code>print_activity("cloudy", 2001)</code>	No feedback (only highlighted in green since this is one of the correct options)
Value 1 indicates first element in list2	Value 1 here indicates an integer number 1

## EXPERIMENTAL DESIGN

There were two conditions in the study; the only difference between the versions of PyKinetic presented to the control and experimental group was that the experimental group additionally received menu-based SE prompts for every LOC completed.

We recruited 47 volunteers from an introductory programming course at University of Canterbury (UC), and 23 volunteers enrolled in an introductory computing course at the Ateneo de Manila University (ADMU). There were 70 participants in total, randomly assigned to two groups: experimental group (with SE prompts) and control group (without SE prompts). The ADMU participants were given free food as compensation. The participants from the University of Canterbury did not receive compensation but were added to a draw for one of four NZ\$50 vouchers. The study was approved by the Human Ethics Committees of both universities.

Each participant joined a group session which lasted for 1.5-2 hours. There were between one and 13 participants per session. At the start of each session, the participants were introduced to the study and provided informed consent. The participants were advised that they could pause or stop at any time during the study. Then, a 15-minute pre-test was administered, and the participants were instructed on how to download and install PyKinetic. The participants used PyKinetic for about an hour. After interacting with PyKinetic, the participants received a 15-minute post-test, and finally, instructions on uninstalling the application. Both pre- and post-tests were completed on paper. Some participants used their own Android smartphones, while we provided phones to other participants. After the post-test, we asked participants who used their own devices for the study to uninstall PyKinetic.

We developed two tests of comparable complexity (given in Appendix). The pre/post-test had a total of eight questions: six conceptual questions (with a maximum of 6 marks) and two procedural questions (2 marks). The conceptual questions were multiple-choice or True/False questions. One procedural question asked the participants to predict the code output (without providing any choices), and the other one was a Parsons problem. Questions with multiple correct answers were marked depending on the options selected. Partial marks were given for selecting correct options, and for not selecting wrong options. Partial marks were deducted for selecting wrong options. This was done to avoid discrepancy for participants who seemed to be guessing answers by selecting all options. Parsons problems from the pre/post-test were marked based on the number of LOCs written in the correct sequential order. One participant wrote a slightly different solution by modifying the incomplete LOCs but received full marks on that question as the solution was correct. There were more conceptual than procedural questions in the pre/post-tests, which was similar proportionally with SE prompts.

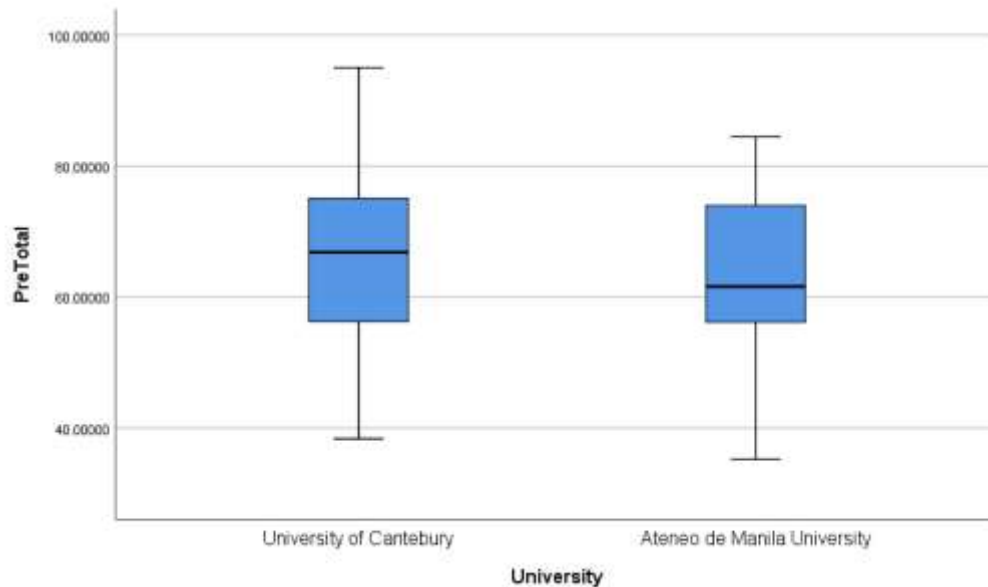
## FINDINGS

We have eliminated data about seven participants due to incomplete logs caused by WiFi connection problems. The paper presents analyses performed on the data collected from the remaining 63 participants. As we had participants from two universities, we compared their pre-test scores. There was no significant difference on the pre-test scores of the two populations (Table 5). Therefore, the two populations had comparable levels of pre-existing knowledge. Figure 4 shows the box plots of pre-test scores from both universities.

**Table 5.** Mean pre-test scores (standard deviations in parentheses) of the two populations of participants

	UC (42 students)	ADMU (21 students)
Pre-test %	66.73 (13.05)	63.40 (13.53)

As the data was not normally distributed, we used non-parametric tests for all reported analyses. We calculated the normalized gain using two formulas. When the learning gain was positive, we calculated the quotient of gain (post-test score – pre-test score) and (100 – pre-test score). However, when the learning gain was negative, we calculated the quotient of gain and the pre-test score (Marx and Cummings, 2007).



**Fig. 4.** Box plots for pre-test total scores of the two populations of participants

On average, the experimental group participants spent 48 minutes using PyKinetic (sd = 13.1), while control group learners spent slightly less time (42 minutes, sd = 14.4). There was no significant difference on the total interaction time between the two conditions. However, there was a significant difference ( $U = 340$ ,  $p = .035$ ) between the two conditions on average time spent per problem, with the control group participants being faster (mean = 3.22 minutes, sd = 1.18) compared to the experimental group (mean = 4 minutes, sd = 1.52). This was expected, as they did not have SE prompts. Only 38 participants finished all problems (11 from experimental and 27 from the control group). Out of fifteen problems, the experimental group participants completed 13.2 problems (sd= 1.9), which was significantly less ( $U = 701.5$ ,  $p = .001$ ) than the control group (14.4 problems, sd=1.5).

We used the paired Wilcoxon Signed Ranks test to examine hypothesis H1. Table 6 reports the pre/post-test scores for the two groups on all questions, and separately on conceptual/procedural questions. Participants from both groups improved their scores significantly between the pre- and post-test (the *Improvement on All Questions* row), and on conceptual questions (the *Improvement Conceptual* row), but there was no significant improvement on procedural questions only. These results provide evidence to accept our first hypothesis H1, which was that PyKinetic helped improve Python skills of the participants. The fact that the two groups improved on conceptual questions but not on procedural questions might be explained by the higher proportion of conceptual vs. procedural in the SE prompts. However, further investigation is needed to prove this speculation. Both groups had a positive Cohen’s d effect size, but the effect size was higher for the experimental group.

**Table 6.** Pre- and post-test scores in % (\* denotes significance at  $p < .05$ )

	<b>Experimental (29)</b> Mean (sd)	<b>Control (34)</b> Mean (sd)	<b>U, p</b>
Pre-test	65.22 (15.31)	65.96 (11.33)	ns
Post-test	77.91 (13.26)	72.60 (12.88)	ns
Improvement on All Questions	W = 379, p = .000*	W = 483, p = .002*	
Cohen’s d for All Questions	d = .89	d = .55	
Normalized Gain All Questions	34.73 (32.34)	20.42 (25.28)	U = 348, p = .046*
Pre-test Conceptual	62.19 (16.42)	63.24 (13.50)	ns
Post-test Conceptual	78.06 (15.41)	71.19 (15.15)	ns
Improvement Conceptual	W = 375, p = .001*	W = 474.5, p = .002*	
Cohen’s d for Conceptual	d = 1.00	d = 0.55	
Normalized Gain Conceptual	41.61 (38.84)	22.08 (30.94)	U = 334, p = .028*
Pre-test Procedural	74.31 (23.36)	74.99 (19.79)	ns
Post-test Procedural	77.45 (16.46)	76.84 (19.50)	ns
Improvement Procedural	ns	ns	
Cohen’s d for Procedural	d = 0.16	d = 0.09	
Normalized Gain Procedural	21.99 (47.34)	29.54 (48.15)	ns

To answer H2, we used Mann Whitney U test for checking significant differences between the groups (Table 6). There was no significant difference on the pre-test scores. Although the experimental group achieved higher marks in the post-test, the difference was not statistically significant. Similarly, participants in the experimental group earned higher marks on the post-test on conceptual questions, however this was not statistically significant either. The experimental group had significantly higher normalized gains for all questions (U = 348, p = .046). Furthermore, the normalized gain for conceptual questions of those who self-explained was also significantly higher (U = 334, p = .028). However, both groups had comparable normalized gains for procedural questions. We also calculated the Cohen’s d effect size of each group. The experimental group had almost double effect sizes compared to the control group on all questions, conceptual, and procedural questions. The Cohen’s d effect size for the normalized gains between the groups is 0.493, indicating a moderate positive effect. The Cohen’s d effect size for the normalized gains on conceptual questions also indicates a moderate positive effect (d = 0.556). On the other hand, for procedural questions between groups the effect size is a weak negative effect (d = -0.158). These findings are enough evidence to support H2, revealing that participants who self-explained had further learning benefits.

To probe further into the effect of SE prompts on deeper learning, we performed a similar analysis as in (Alevén and Koedinger 2002), who also conducted an evaluation with two versions of their system (with and without SE). We classified pre/post test questions into two categories: *easier-to-guess* and *harder-to-guess*. The

easier-to-guess questions were two True/False questions, and one multiple choice question with a single answer required (3 marks in total). The harder-to-guess questions (5 marks in total) required more knowledge to identify the correct answers. For example, the harder-to-guess questions included an output prediction question, which required the student to analyze the given code and think about its output, as there were no options provided. There were also three multiple-choice questions where the student needed to identify all correct options, and a Parsons problem. We used the Wilcoxon Signed Ranks test and Mann Whitney U test for checking significant differences between the groups. The results in Table 7 show that both groups improved significantly on harder-to-guess questions from pre- to post-test. The experimental group outperformed the control group on the harder-to-guess questions. We also calculated the effect sizes for easier and harder-to-guess questions (Table 7). The Cohen's d effect size for the normalized gains on easier-to-guess questions between groups reveals a weak positive effect ( $d = 0.308$ ). For harder-to-guess questions, the between groups Cohen's d effect size is a weak negative effect ( $d = -0.263$ ).

**Table 7.** Pre- and post-test scores (%) for Easier and Harder to Guess questions

(\* denotes significance at  $p < .05$ )

	<b>Experimental (29)</b> Mean (sd)	<b>Control (34)</b> Mean (sd)	<b>U, p</b>
Pre-test Easier-to-Guess	75.57 (23.14)	80.39 (21.89)	ns
Post-test Easier-to-Guess	89.37 (20.03)	87.99 (21.44)	ns
Improvement on Easier-to-Guess	ns	ns	
Cohen's d on Easier-to-Guess	$d = 0.64$	$d = 0.35$	
Normalized Gain Easier-to-Guess	42.53 (64.29)	24.02 (55.69)	ns
Pre-test Harder-to-Guess	59.01 (14.00)	57.30 (10.86)	ns
Post-test Harder-to-Guess	71.03 (14.95)	63.37 (12.58)	$U = 319, p = .016^*$
Improvement on Harder-to-Guess	$W = 368, p = .001^*$	$W = 472, p = .003^*$	
Cohen's d on Harder-to-Guess	$d = 0.83$	$d = 0.52$	
Normalized Gain Harder-to-Guess	12.02 (46.96)	22.69 (33.08)	ns

To investigate H3 (whether Parsons problems with incomplete LOCs are more beneficial for LP learners), we have sorted all 63 participants based on their pre-test scores. We took the top 25% participants and labelled them as high prior knowledge (HP), the lowest 25% participants and labelled them as low prior knowledge (LP). There were nine LP learners and nine HP learners in the experimental group; and seven LP learners and seven HP learners in the control group. There were no significant differences between pre-test scores of LP learners from the two groups (Table 8). Similarly, the pre-test scores of HP students from experimental and control groups were also not significantly different. Furthermore, we found no significant difference between the time spent per problem by LP learners in experimental group compared to those in control group. Similarly, no significant difference was found between the average time spent per problem by HP students in experimental group compared to those in control group.

**Table 8.** Effect sizes calculated by abilities

<b>Group</b>	<b>Subgroup</b>	<b>Pre-test</b> Mean (sd)	<b>Post-test</b> Mean (sd)	<b>Cohen's d</b>
Experimental	LP (9)	48.09 (7.16)	73.40 (16.90)	$d = 1.95$
	HP (9)	83.22 (5.26)	79.71 (15.01)	$d = -.31$
Control	LP (7)	50.01 (6.74)	61.64 (16.81)	$d = .91$
	HP (7)	79.79 (4.88)	82.77 (6.85)	$d = .50$

As presented in Table 8, there is a substantial difference between the Cohen's *d* effect sizes of LP vs. HP participants in both groups. More importantly, the LP in the experimental group had a more than double effect size than LP in the control group. The results in Table 8 show evidence of H3, that Parsons problems with incomplete LOCs are more beneficial for novice learners, specifically those with low prior knowledge.

**Table 9.** LP Students and HP Students from the Experimental Group (\* denotes significance at  $p < .05$ )

Measure (%)	Experimental LP (9) Mean (sd)	Experimental HP (9) Mean (sd)	U, p
Pre-test	48.09 (7.16)	83.22 (5.26)	U = 0, $p = .000^*$
Post-test	73.40 (16.90)	79.71 (15.01)	ns
Improvement on All Questions	W = 44, $p = .011^*$	ns	
Normalized Gain All Questions	49.14 (33.15)	9.64 (33.20)	U = 66, $p = .024^*$
Pre-test Conceptual	46.85 (13.39)	79.63 (5.98)	U = 1, $p = .000^*$
Post-test Conceptual	73.34 (18.01)	78.95 (18.58)	ns
Improvement Conceptual	W = 42, $p = .021^*$	ns	
Normalized Gain Conceptual	46.89 (41.48)	26.00 (48.59)	ns
Pre-test Procedural	51.82 (23.43)	94.01 (5.48)	U = 0, $p = .000^*$
Post-test Procedural	73.61 (21.95)	81.97 (8.53)	ns
Improvement Procedural	W = 42, $p = .021^*$	W = 3, $p = .021^*$	
Normalized Gain Procedural	49.00 (43.93)	-7.11 (20.71)	U = 68, $p = .014^*$
Pre-test Easier-to-Guess	51.85 (17.57)	96.30 (11.11)	U = 2.5, $p = .000^*$
Post-test Easier-to-Guess	88.89 (23.57)	84.26 (23.73)	ns
Improvement Easier-to-Guess	W = 39.5, $p = .041^*$	ns	
Normalized Gain Easier-to-Guess	77.78 (50.69)	-4.56 (45.48)	U = 68.5, $p = .011^*$
Pre-test Harder-to-Guess	45.84 (7.24)	75.38 (9.84)	U = 0, $p = .000^*$
Post-test Harder-to-Guess	64.11 (21.04)	76.98 (13.05)	ns
Improvement Harder-to-Guess	W = 39, $p = .051$	ns	
Normalized Gain Harder-to-Guess	32.56 (42.19)	14.44 (32.73)	ns

We were also interested in more detailed analyses of the performance of LP learners in each condition. In the experimental condition, the LP learners spent a statistically comparable amount of time per problem compared to HP students (*LP*: 4.04 minutes,  $sd = 2.29$ ; *HP*: 4.22 minutes,  $sd = 1.16$ ). We expected the pre/post-scores of LP and HP students to be significantly different, due to how the two subgroups were selected. Although there was a significant difference on the pre-test scores, the performance of LP learners on the post-test was not significantly different from the performance of HP students (Table 9). There were also no significant differences on the post-test scores for procedural, conceptual, easier or harder to guess questions between LP and HP students. LP students also had a significant improvement their pre-/post-test scores on all questions, conceptual, procedural, and easier-to-guess questions. This is contrary to HP students who did not reveal any significant improvement from any of their pre-/post-test scores. Furthermore, a stronger evidence is that LP students had a significantly higher normalized gain than HP students for all questions, procedural, and easier-to-guess questions. However, the pre-to post-test scores of HP students who self-explained might be due to the ceiling effect. Overall, these results show that SE prompts better support LP than HP students.

**Table 10.** Experimental Group SE Scores (\* denotes significance at  $p < .05$ )

Score (%)	Experimental LP (9) Mean (sd)	Experimental HP (9) Mean (sd)	U, p
SE All Questions	56.68 (14.36)	73.83 (10.05)	U = 11, p = .008*
SE Conceptual Questions	59.28 (13.46)	76.04 (11.02)	U = 13, p = .014*
SE Procedural Questions	50.95 (20.46)	69.79 (15.07)	U = 19, p = .063

The students' answers to SE prompts were marked the same way as the multiple-choice questions in the pre/post-test. The HP students' scores for SE prompts were significantly higher overall, and for conceptual SE prompts, but not for the procedural SE prompts (Table 10).

Looking at the control condition, the LP learners spent similar amount of time per problem compared to HP students from the same group (*LP*: 3.24 minutes,  $sd = 1.30$ ; *HP*: 2.87 minutes,  $sd = 1.03$ ). Contrary to results of the experimental group, the HP students scored significantly higher both on the pre- and the post-test in comparison to LP students (Table 11), apart from the post-test score on easier-to-guess questions. Moreover, a striking difference when compared to results of the experimental group is that LP students in the control group only improved significantly on pre- to post-test scores on harder-to-guess questions (only set of questions where LP students in experimental group did not significantly improve). Interestingly, the HP students in the control group improved significantly in procedural questions and in harder-to-guess questions which again is contrary to those in the experimental condition (specifically for HP students). When normalized gains of LP and HP in the control group were compared, only the normalized gain on procedural questions was significantly different, where HP's gain was higher.

**Table 11.** LP Students and HP Students from the Control Group (\* denotes significance at  $p < .05$ )

Measure	Control LP (7) Mean (sd)	Control HP (7) Mean (sd)	U, p
Pre-test	50.01 (6.74)	79.79 (4.88)	U = 0, p = .000*
Post-test	61.64 (16.81)	82.77 (6.85)	U = 2.5, p = .002*
Improvement on All Questions	ns	ns	
Normalized Gain All Questions	22.48 (34.97)	19.58 (19.69)	ns
Pre-test Conceptual	46.83 (11.26)	76.63 (7.11)	U = 0, p = .000*
Post-test Conceptual	58.77 (21.11)	78.21 (7.74)	U = 7.5, p = .026*
Improvement Conceptual	ns	ns	
Normalized Gain Conceptual	20.00 (45.07)	13.86 (18.85)	ns
Pre-test Procedural	63.73 (7.40)	89.28 (13.95)	U = 4.5, p = .007*
Post-test Procedural	70.24 (14.33)	96.43 (9.45)	U = 2, p = .002*
Improvement Procedural	ns	W = 21, p = .026*	
Normalized Gain Procedural	20.43 (37.54)	76.86 (41.02)	U = 6, p = .017*
Pre-test Easier-to-Guess	57.14 (16.27)	100 (0)	U = 0, p = .000*
Post-test Easier-to-Guess	76.19 (37.09)	94.05 (12.47)	ns
Improvement Easier-to-Guess	ns	ns	
Normalized Gain Easier-to-Guess	50.00 (76.38)	-5.86 (12.34)	ns
Pre-test Harder-to-Guess	45.73 (3.41)	67.66 (7.80)	U = 0, p = .000*
Post-test Harder-to-Guess	52.91 (8.17)	76.00 (10.01)	U = 1.5, p = .001*
Improvement Harder-to-Guess	W = 27, p = .028*	W = 27, p = .028*	
Normalized Gain Harder-to-Guess	13.14 (14.89)	27.14 (24.42)	ns



## DISCUSSION

Our results demonstrate the effectiveness of PyKinetic in supporting learning for introductory programming students. Although the participants only interacted with the tutor for roughly 45 minutes, our results still revealed positive learning effects.

One of the contributions of our research is the innovative design of Parsons problems, which contained four crucial differences compared to most other related work. Firstly, Parsons problems in PyKinetic contain only one area acting as both a problem and solution space. As discussed in the Related Work section, in other implementations of Parsons problems learners drag blocks of code from the problem area onto the solution area. Although there are some implementations that also allow learners to rearrange blocks of code within the solution area like in js-parsons (Helminen et al., 2012), the main difference lies on the initial state of the problems. Combining the problem and solution area makes better use of the space in smartphones which allows for longer problems. However, having only one area means that learners might be overwhelmed because all the LOCs are displayed at once in the same space, and learners need to mentally separate their solution from the rest of the LOCs. We recognize that combining the problem and solution area might make it difficult for learners to keep track of the LOCs they have moved. However, we did not observe this to be an issue in our study, probably because we limited the length of the problems to have maximum of 16 LOCs. Secondly, since we designed PyKinetic for novice learners, we provided scaffolding for indentations, contrary to other implementations of Parsons problems (Parsons and Haden 2006; Ericson, Margulieux and Rick 2017; Ihantola and Karavirta 2011; Karavirta, Helminen and Ihantola 2012; Ihantola, Helminen and Karavirta 2013; Kumar, 2018; Harms, Chen, and Kelleher 2016). Thirdly, our design required students to move individual LOCs like Garner (2007) and Kumar (2018), instead of moving blocks of code; thus encouraging students to think about each individual LOC. Lastly, we also introduced incomplete LOCs to further support code writing skills (i.e. procedural knowledge). We acknowledge that acquisition of both procedural and conceptual knowledge is important for novice learners. Therefore, we have introduced menu-based SE prompts for every incomplete LOC, to provide support for conceptual knowledge. The combination of Parsons problems and SE prompts is also one of our main contributions, as this has not been done earlier to the best of our knowledge. We stress that our design of Parsons problems combined SE prompts as an entity is our main contribution rather than the specific differences of our implementation with other work.

Self-explanation has previously been proven effective in many domains, implemented on personal computers, and we found that SE is also beneficial when learning programming in a mobile tutor. We are not aware of other evaluations of menu-based SE on smartphones to the best of our knowledge. Thus, another contribution is designing and evaluating SE on smartphones. Furthermore, we found that SE prompts are effective when combined with activities like Parsons problems, which are less demanding than problem solving but more difficult than studying worked examples.

In our study, the experimental group participants had to respond to all SE prompts. We did not allow the students to skip the SE prompts to support possible missing gaps in the learners' knowledge regardless of their abilities. We suspect that if we had allowed participants to skip the SE prompts, poorer students would probably choose to not attempt the prompts, thus losing chances to deepen knowledge. Aleven and Koedinger (2000) reported that students did not always follow through their tutor's SE prompts.

We presented evidence showing that menu-based SE prompts further improved students' learning. Experimental group participants completed less problems on average, with most not completing the last two problems (i.e. the most difficult problems). Despite completing fewer problems, the experimental group still learned more in comparison to the control group. Students who self-explained had a significantly higher normalized gain for all questions than those who did not. The Cohen's  $d$  effect size on the treatment revealed a moderate effect ( $d = 0.493$ ).

Our findings revealed that students from both groups, regardless of whether they solved SE prompts, improved more on conceptual questions than procedural questions. However, we do not have enough evidence to show that Parsons problems enhance conceptual knowledge more than procedural knowledge. Therefore, further research is needed to verify the specific impact that Parsons problems have on harnessing conceptual and procedural knowledge.

The experimental group had a significantly higher normalized gain on conceptual questions in comparison to the control group. Moreover, a moderate positive effect size was found for conceptual questions ( $d = 0.556$ ). On the contrary, procedural questions yielded a weak negative effect ( $d = -0.158$ ). We have enough evidence to show that SE prompts enhance conceptual knowledge. Based on our results, learners who self-explained performed significantly better on conceptual questions than those who did not. Our results are supported by literature showing SE enhances conceptual knowledge (Chi et al. 1989; Chi et al., 1994; Ferguson-Hessler and de Jong, 1990; Najar, Mitrovic, and McLaren, 2016). Furthermore, we have provided evidence that this also holds for learning with a smartphone tutor. Thus, combining Parsons problems with SE prompts helped learners in the experimental group to perform better on conceptual questions. However, we did not find any evidence that self-explaining is beneficial for enhancing procedural knowledge.

Our results verified that Parsons problems are suitable for novice learners, specifically for those with low prior knowledge. The reason behind this is most likely the amount of scaffolding provided in Parsons problems, as they contain correct syntactic structures. Morrison et al. (2016) also have similar insights that Parsons problems require lower cognitive load than other activities like code writing. Students with high prior knowledge are usually well-versed with syntax compared to those with low prior knowledge, so they often do not require scaffolding for syntax. We also found distinctive evidence that PyKinetic was most effective for LP students who self-explained, which was expected, as weaker students often find it difficult to self-reflect and fill gaps in their own knowledge (Chi et al. 1989). LP learners who self-explained had a notably high Cohen's  $d$  effect size ( $d=1.95$ ), more than double the effect size for LP students in the control group ( $d = .91$ ). Furthermore, LP learners who self-explained had reached the performance of HP students on the post-test. On the contrary, although LP students in the control group improved from their pre- to post-test scores, their post-test scores were still significantly lower in comparison to the HP students from the same group.

A possible explanation for LP students learning more with Parsons problems than HP students is a ceiling effect. It is probable that our HP participants particularly in the experimental group might have shown a ceiling effect. However, in the control group HP learners had significant improvement on procedural questions despite having high pre-test scores. Furthermore, it is thought-provoking that HP learners in the control group acquired significantly higher normalized gains on procedural questions compared to LP students. Due to low numbers in these subgroups, we do not propose that HP students are better without SE. However, we are inclined to suggest that this might be possible.

We do not postulate that a ceiling effect is an accurate explanation for LP students benefitting more than HP students when solving Parsons problems (especially with SE prompts). This is due to results from our other study (Fabic, Mitrovic and Neshatian, 2018b) wherein we evaluated a different version of PyKinetic containing output prediction and debugging problems without Parsons problems. In this study, we have found the opposite outcome, where LP students had negligible learning effects, but HP students had significant learning benefits. Therefore, we have reasons to believe that Parsons problems are indeed more suitable for LP students, while output prediction and debugging problems might be more suitable for HP students. Our findings suggest that this is due to the hierarchy of programming skills. In our other study (Fabic, Mitrovic and Neshatian, 2018b), LP students had difficulties to learn higher order of programming skills (i.e. debugging) since they have not yet mastered code writing, which is an essential skill for debugging. Therefore, LP students did not get significant pedagogical benefits with practicing debugging exercises compared to HP students.

Although learning effects were positive, there are certain aspects of the SE prompts which could be improved. We have chosen to utilize a mixture of questions written in a positive or negative manner, to increase the difficulty of the questions. However, during the study, some of the participants did not read the questions properly and mistakenly answered some SE prompts. For example, a question which had "Select all INCORRECT statements...", confused some participants, where they instead selected all *correct* statements. This issue may or may not be concerning, as we have observed that these questions compelled most participants to be more cautious after this occurrence. Some participants were observed to have completed the incomplete LOC first before reading the rest of the problem. This made the SE prompts to appear more difficult for them, as answering some of the prompts requires students to understand the entire code, not just the incomplete line. It may have helped to display

a message advising learners to rearrange the LOCs first, before filling in the missing keyword/s for the incomplete LOC.

The goals of the evaluation study presented in this paper were to evaluate three hypotheses. The findings supported our first hypothesis H1, with both groups having improved significantly from the pre- to the post-test. Both groups also improved significantly on the harder-to-guess questions. We also showed enough evidence to support our second hypothesis H2, with a moderate effect size, showing that the participants who self-explained, and their normalized gains were also significantly higher than those in the control group. Moreover, the evidence was more distinctive with LP learners. We have also found enough evidence to accept our hypothesis H3, that LP students would learn more than HP students. LP learners from both treatments have learned significantly more than HP students. However, LP students who self-explained benefitted the most, as they achieved similar scores to those of the HP students in the post-test and had a very high Cohen's  $d$  effect size.

We observed that participants were highly engaged, possibly because most of them have not experienced learning Python on smartphones before. Some participants also asked if they could download PyKinetic and keep it to continue learning. Although the participants were informed that they were free to stop the session whenever they wanted, all participants stayed until the time was up. Some participants commented that they wished they could stay longer and finish all the problems in the tutor. On the downside, as participation in the study was voluntary, this may have affected their performance on the pre- and post-tests. Participants may or may not be motivated in answering the exercises as they were informed that they will be compensated similarly regardless of their performance.

## CONCLUSIONS AND FUTURE WORK

A major contribution of our research is in the pedagogically-guided design of our variant of Parsons problems which is unique as an entity. Those activities included 1) one area acting as both problem and solution areas; 2) scaffolding for indentations; 3) blocks of code each with single LOCs; and 4) incomplete LOCs with menu-based SE prompts. There is some evidence that Parsons problems are positively correlated with code writing (Denny et al., 2008); a skill considered as procedural knowledge in programming. Therefore, we have added incomplete LOCs to further support the acquisition of procedural knowledge. We also introduced menu-based SE prompts for every completed LOC to support the advancement of conceptual knowledge. These SE prompts were proven to be effective in PyKinetic which contains puzzle-like exercises, consistent with work by Johnson and Mayer (2010). Our research revealed that menu-based SE prompts are also effective on a mobile platform. We have also addressed a research gap in the area of self-explanation by combining it with an activity that is more challenging than learning with worked examples, but less demanding than problem solving. More specifically, we found that SE is also effective when combined with activities such as Parsons problems. The evaluation of a combination of Parsons problems and SE, as well as menu-based SE prompts on smartphones, have not been done before to the best of our knowledge. Further research on the effectiveness of other types of SE prompts in mobile tutors is needed.

One limitation of our study was the larger proportion of conceptual vs. procedural questions in the SE prompts, pre-test and post-test. This is not alarming, as the proportions in all three were consistently similar. However, the results may have been different with a different proportion of conceptual vs. procedural questions. Another limitation on our study was the session length. Longer sessions may be considered in our future evaluations, as well as a delayed/transfer test, which may yield more interesting results. The number of participants could have also been improved. Since our study was on a mobile device, running controlled experiments was not as straight-forward as running it on personal computers. We found that the WiFi connection on some of the phones were not always reliable, which led to some data loss. We plan to conduct future studies using only our development phones, to be able to save the data in two places (on the server sent through WiFi, and on the smartphones for backup). Another avenue for future work is adding game features to PyKinetic, in order to increase student motivation.

PyKinetic was designed specifically for novice programmers. Our findings support this, and more specifically revealing that PyKinetic is more beneficial to LP students than HP students. Our future work includes adding more problems and developing other kinds of activities for PyKinetic (Fabic, Mitrovic and Neshatian 2017a). We also aim to investigate activities most effective in PyKinetic for participants with different abilities. Lastly, we endeavor to develop an adaptive version of PyKinetic with personalized problem selection (Fabic, Mitrovic and Neshatian 2018a).

## Acknowledgements

We thank all participants as well as our colleagues who helped in administering the study. Special thanks to Professor Ma. Mercedes T. Rodrigo and her team from the Ateneo de Manila University.

## References

- Aleven, V., & Koedinger, K. R. (2000). The need for tutorial dialog to support self-explanation. In *Building dialogue systems for tutorial applications, papers of the 2000 AAAI Fall Symposium* (pp. 65-73).
- Aleven, V. A., & Koedinger, K. R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based Cognitive Tutor. *Cognitive science*, 26(2), 147-179.
- Aleven, V., Ogan, A., Popescu, O., Torrey, C., & Koedinger, K. (2004). Evaluating the effectiveness of a tutorial dialogue system for self-explanation. In *International Conference on Intelligent Tutoring Systems* (pp. 443-454). Springer, Berlin, Heidelberg.
- Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological review*, 89(4):369-406
- Berthold, K., Eysink, T. H., & Renkl, A. (2009). Assisting self-explanation prompts are more effective than open prompts when learning with multiple representations. *Instructional Science*, 37(4), 345-363.
- Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: A case study. *Computer Applications in Engineering Education*, 21(S1), E41- E50.
- Chi, M. T. (2000). Self-explaining expository texts: The dual processes of generating inferences and repairing mental models. *Advances in instructional psychology*, 5, 161-238.
- Chi, M. T., & Wylie, R. (2014). The ICAP framework: Linking cognitive engagement to active learning outcomes. *Educational Psychologist*, 49(4), 219-243.
- Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, 13(2), 145-182.
- Chi, M. T., De Leeuw, N., Chiu, M. H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive science*, 18(3), 439-477.
- Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *Proc. 4<sup>th</sup> Int. workshop on computing education research* (pp. 113-124). ACM.
- Ericson, B. J., Margulieux, L. E., & Rick, J. (2017). Solving Parsons problems versus fixing and writing code. In *Proc. 17<sup>th</sup> Koli Calling Int. Conf. Computing Education Research* (pp. 20-29). ACM.
- Fabic, G., Mitrovic, A., & Neshatian, K. (2016a). Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts. *Proc. 13<sup>th</sup> Int. Conf. Intelligent Tutoring Systems*, (pp. 447-448). Springer.
- Fabic, G., Mitrovic, A., & Neshatian, K. (2016b) Investigating strategies used by novice and expert users to solve Parson's problem in a mobile Python tutor. *Proc. 9<sup>th</sup> Workshop on Technology Enhanced Learning by Posing/Solving Problems/Questions PQTEL 2016*, pp. 434-444, APSCE.
- Fabic, G., Mitrovic A., & Neshatian, K. (2017a) A comparison of different types of learning activities in a mobile Python tutor. *Proc. 25<sup>th</sup> International Conference on Computers in Education*, (pp. 604-613).

- Fabic, G., Mitrovic, A., & Neshatian, K. (2017b). Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python Tutor. In: *E. Andre, R. Baker, X. Hu, M. Rodrigo, B. du Boulay (Eds.), Proc. 18<sup>th</sup> International Conference on Artificial Intelligence in Education* (pp. 498-501). Springer, Cham.
- Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018a). Adaptive Problem Selection in a Mobile Python Tutor. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization* (pp. 269-274). ACM.
- Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2018b). Investigating the effects of learning activities in a mobile Python tutor for targeting multiple coding skills. *Research and practice in technology enhanced learning*, 13(1), 23.
- Ferguson-Hessler, M. G., & de Jong, T. (1990). Studying physics texts: Differences in study processes between good and poor performers. *Cognition and Instruction*, 7(1), 41-54.
- Gadgil, S., Nokes-Malach, T. J., & Chi, M. T. (2012). Effectiveness of holistic mental model confrontation in driving conceptual change. *Learning and Instruction*, 22, 47-61.
- Garner, S. (2007). An exploration of how a technology-facilitated part-complete solution method supports the learning of computer programming. *Issues in Informing Science & Information Technology*, 4, 491-502.
- Grandl, M., Ebner, M., Slany, W., & Janisch, S. (2018). It's in your pocket: A MOOC about programming for kids and the role of OER in teaching and learning contexts. In *Conference Proceeding Open Education Global Conference*.
- Harms, K. J., Chen, J., & Kelleher, C. (2016). Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proc. ACM Conference on International Computing Education Research* (pp. 241-250). ACM.
- Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems? An analysis of interaction traces. In *Proceedings of the ninth annual international conference on International computing education research* (pp. 119-126). ACM.
- Hosseini, R. (2018). Program Construction Examples in Computer Science Education: From Static Text to Adaptive and Engaging Learning Technology (Doctoral dissertation, University of Pittsburgh).
- Hsu, C. Y., Tsai, C. C., & Wang, H. Y. (2012). Facilitating third graders' acquisition of scientific concepts through digital game-based learning: The effects of self-explanation principles. *The Asia-Pacific Education Researcher*, 21(1), 71-82.
- Hürst, W., Lauer, T., & Nold, E. (2007, March). A study of algorithm animations on mobile devices. In *ACM SIGCSE Bulletin* (Vol. 39, No. 1, pp. 160-164). ACM.
- Ihantola, P., & Karavirta, V. (2011). Two-dimensional Parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education*, 10.
- Ihantola, P., Helminen, J., & Karavirta, V. (2013). How to study programming on mobile touch devices: interactive Python code exercises. In *Proc. 13<sup>th</sup> Koli Calling Int. Conf. Computing Education Research* (pp. 51-58). ACM.
- Johnson, C. I., & Mayer, R. E. (2010). Applying the self-explanation principle to multimedia learning in a computer-based game-like environment. *Computers in Human Behavior*, 26(6), 1246-1252.
- Karavirta, V., Helminen, J., & Ihantola, P. (2012). A mobile learning application for Parsons problems with automatic feedback. In *Proc. 12<sup>th</sup> Koli Calling Int. Conf. Computing Education Research* (pp. 11-18). ACM.
- Kumar, A. N. (2018). Epplets: A Tool for Solving Parsons Puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 527-532). ACM.
- Kwon, K., Kumalasari, C. D., & Howland, J. L. (2011). Self-explanation prompts on problem-solving performance in an interactive learning environment. *Journal of Interactive Online Learning*, 10(2), 96-112.
- Lister, R., Clear, T., Bouvier, D. J., et al. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, 41(4), 156-173.
- Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008, September). Relationships between reading, tracing and writing skills in introductory programming. In *Proc. 4<sup>th</sup> Int. workshop on computing education research* (pp. 101-112). ACM.

- Marx, J. D., and K. Cummings. (2007). "Normalized Change." *American Journal of Physics* 75 (1): 87. doi:10.1119/1.2372468.
- Mbogo, C., Blake, E., & Suleman, H. (2016). Design and use of static scaffolding techniques to support Java programming on a mobile phone. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 314-319). ACM.
- Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). Subgoals Help Students Solve Parsons Problems. In *Proc. 47th ACM Technical Symposium on Computing Science Education* (pp. 42-47). ACM.
- Najar, A. S., Mitrovic, A. & McLaren, B. M. (2016). Learning with Intelligent Tutors and Worked Examples: Selecting learning activities adaptively leads to better learning outcomes than a fixed curriculum. *User Modeling and User-Adapted Interaction*, 26, 459-491.
- O'Neil, H. F., Chung, G. K., Kerr, D., Vendlinski, T. P., Buschang, R. E., & Mayer, R. E. (2014). Adding self-explanation prompts to an educational computer game. *Computers in Human Behavior*, 30, 23-28.
- Oyelere, S. S., Suhonen, J., Wajiga, G. M., & Sutinen, E. (2018). Design, development, and evaluation of a mobile learning application for computing education. *Education and Information Technologies*, 23(1), 467-495.
- Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education – vol. 52* (pp. 157-163). Australian Computer Society, Inc.
- Plass, J. L., Moreno, R., & Brünken, R. (2010). *Cognitive load theory*. Cambridge University Press.
- Rau, M. A., Aleven, V., & Rummel, N. (2015). Successful learning with multiple graphical representations and self-explanation prompts. *Journal of Educational Psychology*, 107(1), 30.
- Someren, M. V., Barnard, Y. F., & Sandberg, J. A. (1994). *The think aloud method: a practical approach to modelling cognitive processes*. Academic Press.
- van der Meij, J., & de Jong, T. (2011). The effects of directive self-explanation prompts to support active processing of multiple representations in a simulation-based learning environment. *Journal of Computer Assisted Learning*, 27, 411– 423
- Vinay, S., Vaseekharan, M., & Mohamedally, D. (2013). RoboRun: A gamification approach to control flow learning for young students with TouchDevelop. *arXiv preprint arXiv:1310.0810*.
- Wen, C., & Zhang, J. (2015). Design of a microlecture mobile learning system based on smartphone and web platforms. *IEEE Transactions on Education*, 58(3), 203-207.
- Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3):17-22
- Wylie, R., & Chi, M.T.H. (2014) The Self-Explanation Principle in Multimedia Learning, in Mayer, R.E. (ed.) *The Cambridge Handbook of Multimedia Learning*. Cambridge: Cambridge University Press, pp. 413–432.

## APPENDIX

### Pre-test

1. Circle all **syntactically valid** Python strings from the following:
  - a. 'Herbert the Heffalump'
  - b. "He said "Hi!" to me"
  - c. ""IAm\$a\$tr!nG, d0c\$tr!nG...^\_""
  - d. "'Oh no!'", He exclaimed.'
  - e. 'It\'s raining and pouring, the sun\'s never returning.'
  - f. "One line/nTwo lines/nThree lines/nFour lines/n"
2. The following code snippet produces an error. Why this is the case? (Circle the answer)

```
a = "b"
a = 49
print("qr" + a + "st")
```

- a. "qr" and "st" are not variables. Only variables can use '+' operator.
  - b. You cannot use the '+' operator on string values.
  - c. The '+' operator only works on values with the same type. i.e. variable a in this case is an int while the others are strings.
  - d. Pylint does not like variable a since its name is too short.
3. Circle all that apply about if conditional statements (compared to elif conditional statements).
  - a. if statements are normally used when conditions are **mutually inclusive** i.e. if -> if -> else (the two if statements are mutually inclusive)
  - b. if statements are normally used when conditions are **mutually exclusive** i.e. if -> if -> else (the two if statements are mutually exclusive)
  - c. if statements are used at the start of a set of conditional clauses
  - d. if statements are used at the end of a set of conditional clauses
4. What is the output of the following code snippet:

```
number = 42
message = "hello"
if number > 16:
    print("I am greater than 16.")
if number > 35 and message == "HELLO":
    print("I am greater than 35.")
elif number >= 40 or message == "hello":
    print("Comparing to 40.")
else:
    print("Today is a good day.")
```

5. `data` is a variable with a list value and `n` has a value of 1. Circle all code statements that will give variable `my_var` a non-empty list value. Assume that `data` contains at least two elements.
- `my_var = data[n]`
  - `my_var = data[0:n]`
  - `my_var = data[-1]`
  - `my_var = data[:-1]`
  - `my_var = data[-1:]`
6. Rearrange the following lines and fill in the missing elements of the function `three_odds()` which takes a list `numbers` as an argument and returns the first three odd elements of the numbers. The test cases must also be rearranged to match the expected output. Note: Assume that the input contains at least three odd numbers. Write the line numbers to answer.

Expected Output:

```
[5, 9, 11]
[11, 39, 1]
[11, 9, 3]
```

Code:

```
1 print(three_odds([2,4,10,11,9,3,29]))
2 def three_odds(numbers):
3     return result
4     count += 1
5     if numbers[count] % 2 != 0:
6     result = []
7 print(three_odds([11,39,10,12,84,1]))
8     result.append(numbers[count])
9     while len(_____) < 3:
10    count = 0
11 print(three_odds([5,9,10,11,39,10]))
12     """Return the first three odd numbers in a list"""
```

7. You need to always specify a counter when using a for loop. i.e. `i += 1`. True or False?
- True                      False
8. A tuple can contain a mix of data types. Therefore, the following is valid:
- ```
(('hello', 32, 93.0), [30, 39, 2984, 39])
```
- True                      False



## Post-test

1. Circle all **syntactically valid** Python variables from the following:
  - a. Variable
  - b. MyTuple
  - c. iAmAvar\$^&
  - d. x\_y
  - e. a1
  - f. 1a
2. The following code snippet produces an error. Why this is the case? (Circle the answer)

```
x = 19
x = [17, 39, 40, 289]
print(20 + x + 60)
```

- a. You cannot use the '+' operator on variables.
  - b. Pylint does not like variable x since its name is too short.
  - c. 20 and 60 are ints, the '+' operator can only be used on strings.
  - d. The value that x is referencing was changed to a list. x has to also be an int to use the '+' operator with 20 and 60.
3. Circle all that apply about elif compared to if conditional statements.
    - a. elif statements do not need a condition when used i.e. elif: is valid
    - b. elif statements requires a condition when used i.e. elif: is **not** valid
    - c. elif statements are normally used when conditions are **mutually inclusive** i.e. if -> elif -> else (the if and elif statements are mutually inclusive)
    - d. elif statements are normally used when conditions are **mutually exclusive** i.e. if -> elif -> else (the if and elif statements are mutually exclusive)
  4. What is the output of the following code snippet:

```
message = "hi"
my_list = [8, 45, 90, 23]

if my_list[-1] > 16:
    print("I am greater than 16.")
if my_list[2] < 20 or message == "hiho":
    print("I am less than 20.")
elif my_list[3] >= 40 and message == "hi":
    print("Comparing to 40.")
else:
    print("Tomorrow was a good day.")
```

Output:

5. Suppose `data = ["hello", "hi", "greetings", "kia ora"]` and `message` has a string value. Circle all code statements that will give variable `my_var` a string value.

- f. `my_var = data[0:1]`
- g. `my_var = data["kia ora"]`
- h. `my_var = data[-1]`
- i. `my_var = data[1][-1]`
- j. `my_var = data[message]`

6. Rearrange the following lines of the function `abs_reversed()` which takes a list `numbers` as an argument and returns the absolute values of the elements in the list `numbers` in a reverse order. The test cases must also be rearranged to match the expected output.

Expected Output:

```
[10, 37, 12, 20, 0]
[12, 11, 10, 29, 40]
[10, 39, 8, 3]
```

Code:

```
1     result = []
2     for i in range(len(numbers)-1, -1, -1):
3     abs_reversed([3,-8,39,10])
4     def abs_reversed(numbers):
5         if numbers[i] < 0:
6     abs_reversed([40,-29,-10,-11,-12])
7         result.append(numbers[i])
8     abs_reversed([0,20,-12,37,-10])
9     """takes a list of numbers and returns its' absolute values in reverse"""
10        result.append(numbers[i] * -1)
11    return result
12    else:
```

7. For loops can be converted to while loops. True or False?
8. A tuple can contain another tuple. However, lists cannot contain another list. True or False?